第1章

ディジタル信号処理に不可欠な乗算器を使いこなす

――HDLによる記述のしかたとFPGA開発ツールの設定

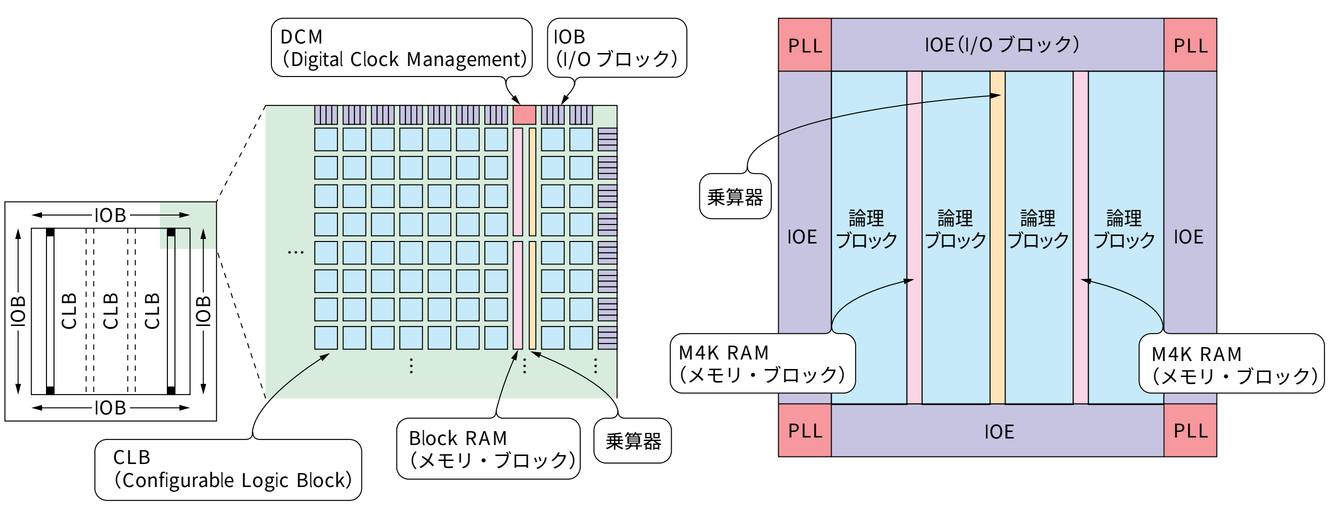
田中良平

ここでは，FPGAにおける乗算器の活用法を解説する．最近のFPGAは，低コスト・ファミリでは18ビット×18ビットの乗算器を，高性能ファミリでは積和演算機能を持つDSPブロックを搭載している．ディジタル信号処理のように，多くの演算を行うアプリケーションの設計では，乗算器を効果的に活用することで，高い性能を実現できるようになる．　（編集部）

最近のFPGA（field programmable gate array）は，単なるプログラミング可能な論理ブロックのかたまりではなくなっています．あらかじめハード・ワイヤード論理で作られた乗算器，クロック管理，メモリなどの機能が組み込まれています（**図1**）．

乗算器は論理ブロックを使えば実現可能です．しかし，演算精度を上げようとすると多くの論理ブロックを消費し，遅延も大きくなってしまいがちです．これに対して，ハード・ワイヤード論理の乗算器であれば，シリコン上で構成できる最高の性能が保証されることになります．アプリケーション回路の設計において，特定の機能についての回路規模や動作速度への影響を気にする必要がなくなることは，大きなメリットになります．あらかじめ組み込まれた機能を活用することは，これからはとても重要になるでしょう．

主要なFPGAが持つ，これらの非論理機能を**表1**にまとめます．例えば，米国Xilinx社のFPGA「Spartan-3ファミリ」には，18ビット×18ビットの演算ができる4～104個の乗算器が搭載されています（**表2**）．「Virtex-4ファミリ」ではこのハード・ワイヤード論理の乗算器に加算器とマル



（a）Spartan-3ファミリの構造　　（b）Cyclone II ファミリの構造

図1　FPGAの構造

（a）は，Xilinx社のSpartan-3ファミリのうちのXC3S200からXC3S2000までの構造を示している．XC3S4000とXC3S5000では，左側のチップ全体の図の点線で示した列の部分（中央のCLBの中）もメモリ・ブロック（Block RAM）になる．XC3S50では，乗算器やBlock RAMは左側の1列だけになる．（b）はAltera社のCyclone IIファミリのEP2C20の構造を示している．

表1　主要なFPGAが持つ非論理機能

メーカ	ファミリ名	乗算器	メモリ	クロック管理
Altera社	Cyclone	なし	M4K RAM　13～64個	PLL　1～2個
	Cyclone II	18ビット×18ビット 13個～150個	M4K RAM　26～250個	PLL　2～4個
	Stratix	DSPブロック 6～22個	M512 RAM　94～767個 M4K RAM　60～364個 M-RAM　1～9個	PLL　6～12個
	Stratix II	DSPブロック 12～96個	M512 RAM　104～930個 M4K RAM　78～768個 M-RAM　0～9個	PLL　6～12個
Xilinx社	Spartan-3	18ビット×18ビット 4～104個	Block RAM　72K～1,872Kビット	DCM　2～4個
	Virtex-II Pro	18ビット×18ビット 12～444個	Block RAM　216K～7,992Kビット	DCM　4～12個
	Virtex-4 LX	DSPブロック 32～96個	Block RAM　864K～6,048Kビット	DCM　4～12個
	Virtex-4 SX	DSPブロック 128～512個	Block RAM　2,304K～5,760Kビット	DCM　4～8個
	Virtex-4 FX	DSPブロック 32～192個	Block RAM　648K～9,936Kビット	DCM　4～20個

DSPブロック ： 積和演算などが可能．Altera社とXilinx社では構造が異なる
M512 RAM ： 512ビット＋パリティのメモリ・ブロック
M4K RAM ： 4Kビット＋パリティのメモリ・ブロック
M-RAM ： 512Kビット＋パリティのメモリ・ブロック
分散RAM ： 論理ブロック内のLUTを使用したメモリ
Block RAM ： 18Kビットのメモリ・ブロック
PLL ： phase-locked loop
DCM ： Digital Clock Management

表2　Spartan-3ファミリの概要

型　名	システム・ゲート	ロジック・セル	CLBアレイ(1CLB＝4スライス) ロウ	カラム	総数	分散RAM(ビット)	Block RAM(ビット)	乗算器	DCM数	最大ユーザI/O数	最大差動I/O数	パッケージ
XC3S50	5万	1,728	16	12	192	12K	72K	4	2	124	56	VQ100，TQ144，PQ208
XC3S200	20万	4,320	24	20	480	30K	216K	12	4	173	76	VQ100，TQ144，PQ208
XC3S400	40万	8,064	32	28	896	56K	288K	16	4	264	116	TQ144，PQ208，FT256
XC3S1000	100万	17,280	48	40	1,920	120K	432K	24	4	391	175	FT256，FG320，FG456，FG676
XC3S1500	150万	29,952	64	52	3,328	208K	576K	32	4	487	221	FG320，FG456，FG676
XC3S2000	200万	46,080	80	64	5,120	320K	720K	40	4	565	270	FG676，FG900
XC3S4000	400万	62,208	96	72	6,912	432K	1,728K	96	4	712	312	FG900，FG1156
XC3S5000	500万	74,880	104	80	8,320	520K	1,872K	104	4	784	344	FG900，FG1156

チプレクサも加わったDSPブロックに進化しています．

乗算器の使いかた

ディジタル信号処理（DSP）アプリケーションにおいては，非常に多くの乗算処理が行われます．乗算回路はどうしても回路規模が大きくなりがちです．そこで，あらかじめFPGAに搭載された乗算器を使いこなせるかどうかが，ディジタル信号処理アプリケーション構築に際してとても重要な要素となります．

FPGAに乗算器が搭載されていることを知りながら，きちんと活用していないケースもあるようです．乗算器はとても簡単に利用できるので，これを機会に積極的に使用してみてください．

● ＊で手軽に記述する

HDLでは，乗算回路はきわめて簡単に記述することができます．

リスト1に18ビット×18ビット乗算回路の記述例を示します．VHDLでは，加算で＋を使うのと同様に＊と記述するだけです．＊を使用するためには，IEEE.std_logic_signedのパッケージ呼び出しが必要です．signedではなく，unsignedにすると，18ビット×18ビットの乗算のために，3個の乗算器を使用してしまいます．

＊で記述した乗算回路を乗算器に割り当てるのはFPGA開発ツールのしごとです．乗算回路の合成としては，ハード・マクロで実装された乗算器を使うほかに，FPGAの基本論理ブロックを構成するLUT（lookup table）で実現する方法もあります．そこで，**図2**に示した論理合成プロパティで，＊で記述した乗算回路をハード・マクロの乗算器に

リスト1　18ビット×18ビット乗算回路のVHDL記述例

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_SIGNED.all;
                                --< UNSIGNEDではないところに注意

entity Multiplier_s is
    generic(
        D_width : integer :=18
      );
    port (
        CLK : in std_logic;
        RST : in std_logic;
        CE  : in std_logic;
        B   : in std_logic_vector(D_width-1 downto 0);
        C   : in std_logic_vector(D_width-1 downto 0);
        A   : out std_logic_vector((D_width*2-1) downto 0)
    );
end Multiplier_s;

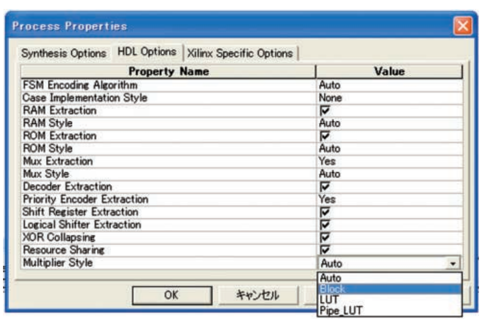
architecture RTL of Multiplier_s is

signal B_node : std_logic_vector(D_width-1 downto 0);
signal C_node : std_logic_vector(D_width-1 downto 0);

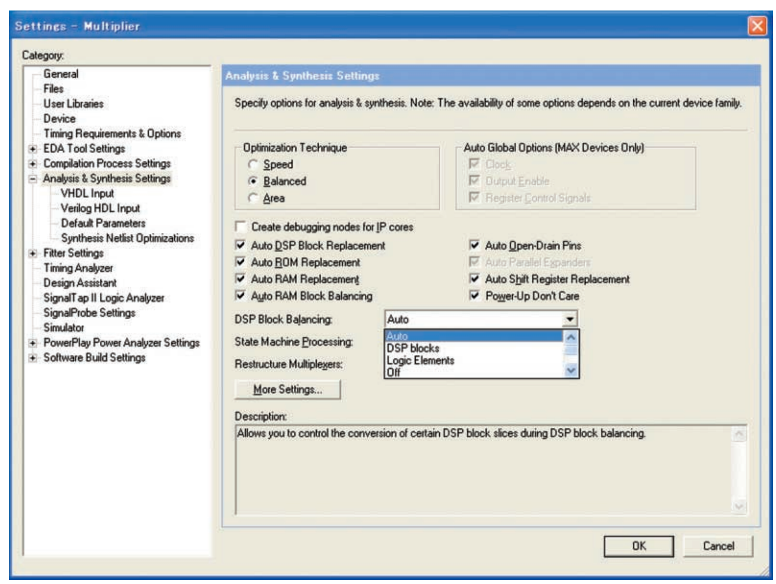
begin

process(CLK, RST) begin
    if(RST = '1') then
        A      <= (others => '0');
        B_node <= (others => '0');
        C_node <= (others => '0');
    elsif(CLK'event and CLK = '1') then
        if(CE = '1') then
            A      <= B_node * C_node;
            B_node <= B;
            C_node <= C;
        end if;
    end if;
end process;

end RTL;
```



(a) ISE 6.3i



(b) Quartus II 4.2

図2
論理合成のプロパティ設定
ISE 6.3iでは，乗算器に実装する場合は「Auto」または「Block」を選択する．Quartus IIでは，「Auto」または「DSP blocks」を選択する．

表3　コンパイル結果の比較

	乗算器(AUTO)	LUT
ISE 6.3i XC3S50VQ100-4	Device utilization summary: Number of External IOBs　75 out of 124　60% Number of LOCed External IOBs　0 out of 75　0% Number of MULT18X18s　1 out of 4　25% Number of BUFGMUXs　1 out of 8　12%	Device utilization summary: Number of External IOBs　75 out of 124　60% Number of LOCed External IOBs　0 OUT OF 75　0% Number of Slices　175 OUT OF 768 22% Number of BUFGMUXs　1 OUT OF 8　12%
Quartus II 4.2 EP2C5Q208C6	Total logic elements　0/4,608(0%) Total registers　72 Total pins　75/142(52%) Total virtual pins　0 Total memory bits　0/199,808(0%) Embedded Multiplier 9-bit elements　2/26(7%) Total PLLs　0/2(0%)	Total logic elements　432/4,608(9%) Total registers　72 Total pins　75/142(52%) Total virtual pins　0 Total memory bits　0/119,808(0%) Embedded Multiplier 9-bit elements　0/26(0%) Total PLLs　0/2(0%)

注：コンパイル結果はツールのバージョンやオプション設定によって変わることがある

割り当てる設定を行う必要があります．Xilinx社のISE 6.3iの場合，Process for Sourceウィンドウの「Synthesize - XST」を右クリックし，「Properties...」を選択します．Process Property画面のHDL Optionタブの中のMultiplier Styleで設定します．米国Altera社のQuartus II 4.2 SP1の場合，「Assignments」→「Settings...」でSettingsウィンドウを開きます．ウィンドウの左側にあるAnalysis & Sysnthesis SettingsのDSP Block Balancingで設定します．

コンパイル（論理合成および配置配線）の結果を**表3**に示します．乗算器が割り当てられたかどうかは，Xilinx社のISE 6.3iならば，Process for Sourceウィンドウの「Implement Design」→「Place & Route」を選択して「Place & Route Report」をダブルクリックすると表示されるレポート・ファイルで確認します．乗算器を使用するように指示した場合は，「Number of MULT18x18s」という項目が出現し，正しく乗算器が使われていることが示されます．乗算器を使用していないときはこの項目が現れません．LUTの使用率なども確認できます．乗算器に割り当てられると，論理ブロックがまったく使われていないこともわかります．

● データ幅や演算式の変更の影響を探る

演算のビット幅を変化させると，必要な規模に応じて最適な数のリソースが割り当てられることになります．デー

リスト2　演算式とデータ幅を変更したVHDLソース・コード

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_SIGNED.all;

entity Multiplier is
    generic(
        D_width : integer :=16
      );
    port (
        CLK : in std_logic;
        RST : in std_logic;
        CE  : in std_logic;
        B   : in std_logic_vector(D_width-1 downto 0);
        C   : in std_logic_vector(D_width-1 downto 0);
        A   : out std_logic_vector((D_width*2-1) downto 0)
    );
end Multiplier;

architecture RTL of Multiplier is

signal B_node : std_logic_vector(D_width-1 downto 0);
signal C_node : std_logic_vector(D_width-1 downto 0);

begin

process(CLK, RST)begin
    if(RST = '1')then
        A      <= (others => '0');
        B_node <= (others => '0');
        C_node <= (others => '0');
    elsif(CLK'event and CLK = '1')then
        if(CE = '1')then
          --- どちらかを選択
            A      <= B_node * C_node * B_node;  --式1
          --A      <= B_node * C_node * B_node *
                                      C_node;  --式2

            B_node <= B;
            C_node <= C;
        end if;
    end if;
end process;

end RTL;
```

表4　データ幅や演算式の変更の影響

	A = B_node * C_node * B_node ;	A = B_node * C_node * B_node * C_node ;
AUTO / BLOCK	Number of MULT18X18s 3 out of 4 75% Number of Slices 8 out of 768 1%	エラー Number of MULT18X18s 5 out of 4 125% XC3S200の場合 Number of MULT18X18s 5 out of 12 41% Number of Slices 17 out of 1920 1%
LUT	Number of Slices 324 out of 76842%	Number of Slices 508 out of 768 66%

（a）D_width : integer : =16固定

	D_width : integer : =18	D_width : integer : =20
AUTO / BLOCK	Number of MULT18X18s 4 out of 4 100% Number of Slices 11 out of 768 1%	エラー Number of MULT18X18s 9 out of 4 225% XC3S200の場合 Number of MULT18X18s 9 out of 12 75% Number of Slices 55 out of 1920 2%
LUT	Number of Slices 409 out of 768 53%	Number of Slices 505 out of 768 65%

（b）A = B_node * C_node * B_node ; 固定

1

タ幅や演算式の変更によってコンパイル結果が変わるようすを見てみましょう．

データ幅を18ビットから変更するためには，**リスト1**の，

```
D_width : integer :=18
```

の部分を修正します．演算式は，

```
A = B_node * C_node;
```

の部分です．

```
A = B_node * C_node * B_node * C_node ;
```

というように，乗算の数を変えていきます．

乗算数を変えたVHDLソース・コードを**リスト2**に，ISE 6.3iによるコンパイル結果を**表4**に示します．3回の乗算を行う回路なら，5個の乗算器を必要とします．使用する乗算器の数は，＊の数と一致しないので注意が必要です．

ターゲット・デバイスがXC3S50の場合，4個の乗算器しか搭載していないため，5個の乗算器を必要とする回路を実装しようとするとエラーになります．しかし，乗算回路をLUTに割り当てるように論理合成ツールに指示すれば，データ幅を16ビットとしたときに実装可能になります．

5個の乗算器を使う回路であれば，4個の乗算器とLUTで実装できそうですが，自動的に合成してはくれないようです．このような場合，乗算器を使う部分を次に説明するコンポーネントで明示して記述し，残りは＊で記述したうえで乗算器の構成をLUTに指定してコンパイルします．

リスト3　コンポーネントを直接記述

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
library UNISIM;
use UNISIM.vcomponents.all;

entity Multiplier_cmp is
    generic(
                D_width : integer :=18
      );
    port (
        CLK : in std_logic;
        RST : in std_logic;
        CE  : in std_logic;
        B   : in std_logic_vector(D_width-1 downto 0);
        C   : in std_logic_vector(D_width-1 downto 0);
        A   : out std_logic_vector((D_width*2-1)
                                                downto 0)
    );
end Multiplier_cmp;

architecture RTL of Multiplier_cmp is

begin
   MULT18X18S_inst : MULT18X18S
   port map (
      P => A,     -- 36-bit multiplier output
      A => B,     -- 18-bit multiplier input
      B => C,     -- 18-bit multiplier input
      C => CLK,   -- Clock input
      CE => CE,   -- Clock enable input
      R => RST    -- Synchronous reset input
);

end RTL;
```

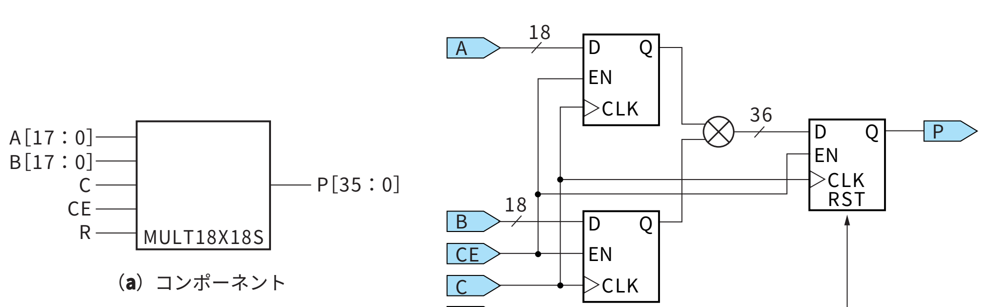
● コンポーネントで記述する

リスト3に，乗算器を明示的に記述する例を示します．**図3**のMULT18X18Sコンポーネントを指定しています．UNISIMのライブラリ宣言と，UNISIM.vcomponentsのパッケージ呼び出しが必要です．

コンパイル結果を**表5**に示します．**リスト1**と同じように，乗算器に割り当てられていることがわかります．**リスト3**のように直接コンポーネントを指定した場合は，**図2**の指定で乗算回路をLUTに割り当てるように指示しても，強制的にハード・マクロが使用されることになります．

表5　MULT18X18Sを使用した場合のコンパイル結果

AUTO/BLOCK	Number of MULT18X18s 1 out of 4 25%
LUT	Number of MULT18X18s 1 out of 4 25% LUT指定を行っても，強制的にMULT18X18sが使われている．



(a) コンポーネント　(b) ブロック図

図3
MULT18X18Sコンポーネントの概要
出力レジスタ付きの18ビット×18ビット乗算器コンポーネントである．

Spartan-3の場合，乗算器は，図4のようにメモリ・ブロックと論理ブロックの間に配置されています．ディジタル信号処理では，演算結果をメモリに貯めておくことがよくあります．適切な配置を考慮して回路を設計することで，性能向上が期待できます．

● IPコア生成ツールでカスタマイズ

乗算器の機能をカスタマイズしたいときには，FPGA開発ツールが持つIPコア生成機能を使用します．Xilinx社のFPGA開発ツールISEであれば，CORE Generatorを使用します（有償版のISEのみ対応している）．Altera社のQuartus IIには，MegaWizardという機能があります．

例えば，符号付き/符号なし演算の選択や入出力レジスタの有無などの選択が可能です．設定した動作に必要な乗算器周辺回路を含めた状態でネットリストが生成されます．また，シミュレーション用のファイルも生成されます．

ここで生成されるネットリストは，論理合成の性能に左右されることなく一定の性能を引き出すことができるため，速度や面積の最適化にたいへん寄与します．筆者の場合，乗算器を使用する場合なら，ほとんどと言ってよいほどこのツールで生成したコンポーネントを使用しています．

Xilinx社のISEの機能であるCORE Generatorによる乗算器の設定手順を図5に示します．すべての設定を行ってから，ISEのテンプレート機能を使用（「Edit」→「Language



図4　乗算器の配置

Spartan-3の例を示している．メモリ・ブロックと論理ブロックの間に配置されている．

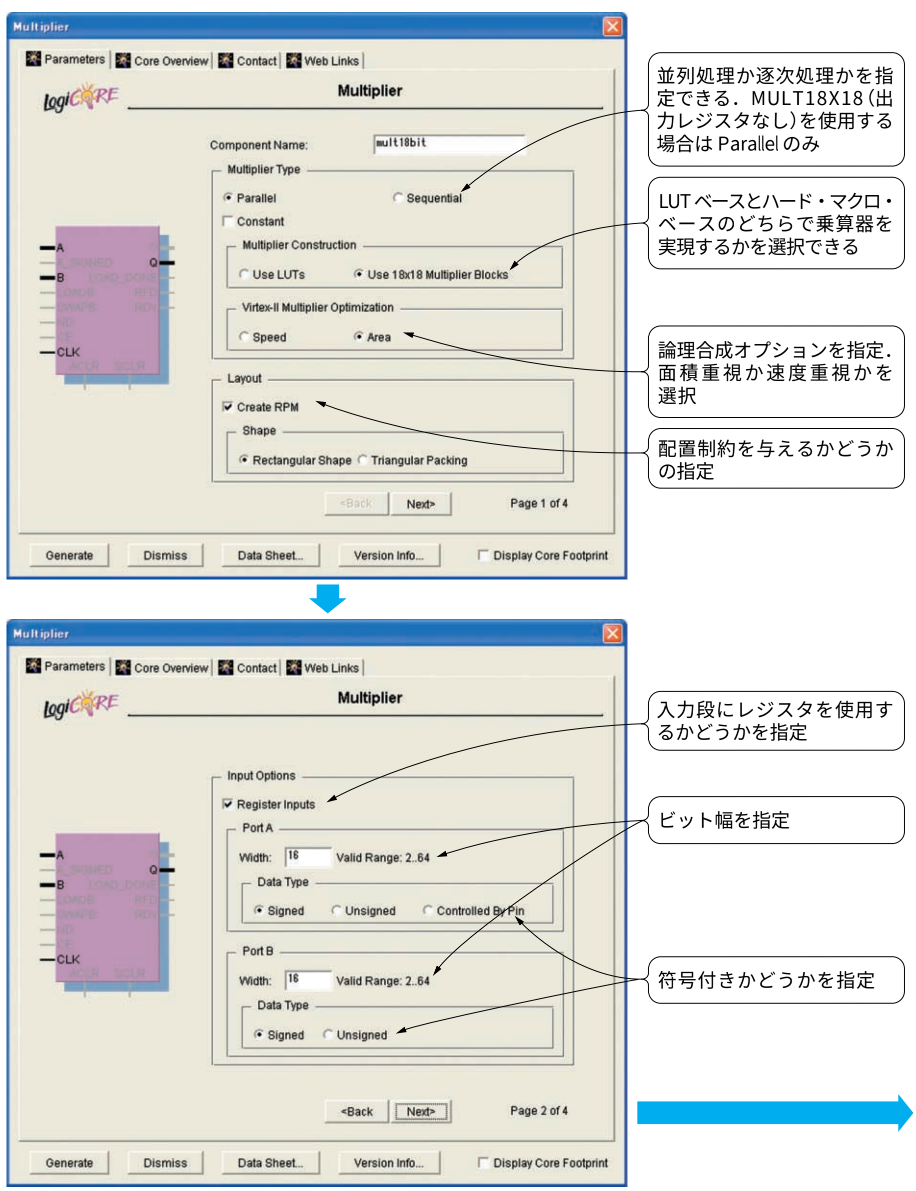
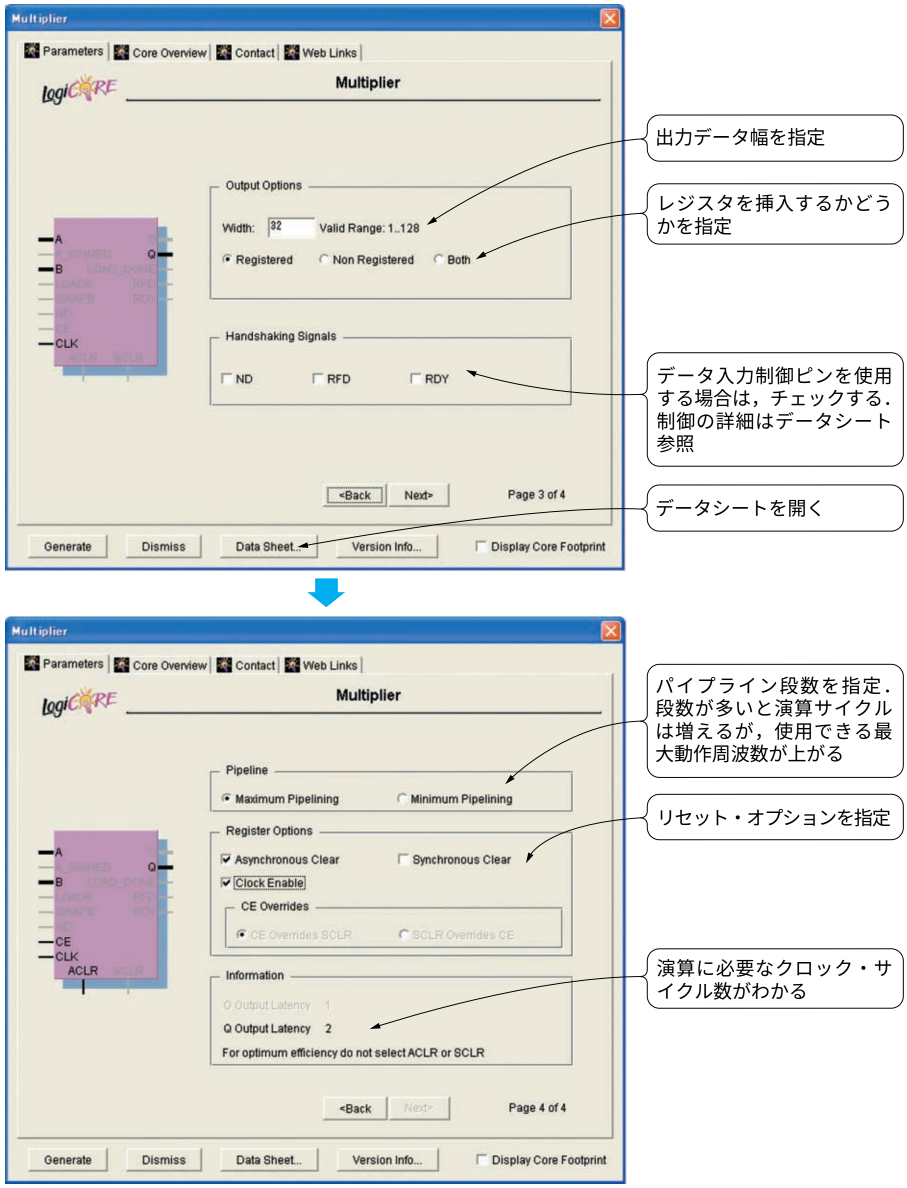


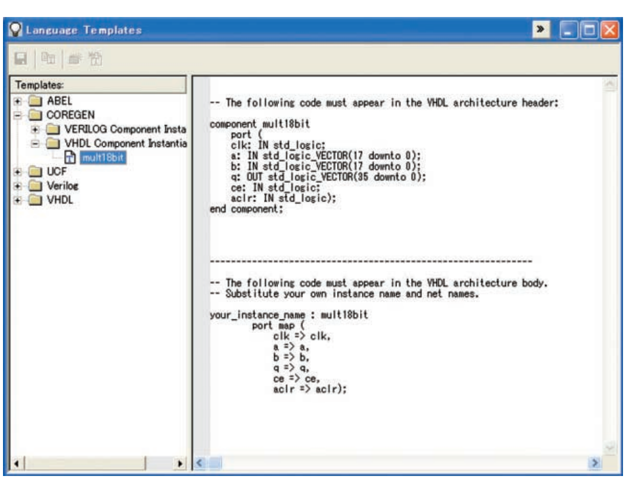
図5
CORE Generatorを使用して乗算器を作成

符号付き/符号なし演算の選択や入出力レジスタの有無などの選択が可能．

図5
CORE Generatorを使用して乗算器を作成（つづき）



▼図6　テンプレートを参考にコンポーネントを挿入
作成した名まえのコンポーネントの記述例を見ることができる．



Template」を選択）すれば，図6の画面になります．COREGENフォルダから作成した名まえのコンポーネントの記述例を見ることができます．あとはリスト3と同様の記述でコンポーネント登録するとよいでしょう．

場合によってはこのテンプレートが表示されないときがありますが，ISEのProcess Viewから「Regenerate Core」を実行することで使用できるようになります．

DSPブロックの使いかた

ディジタル信号処理では，非常に多くの積和演算を行

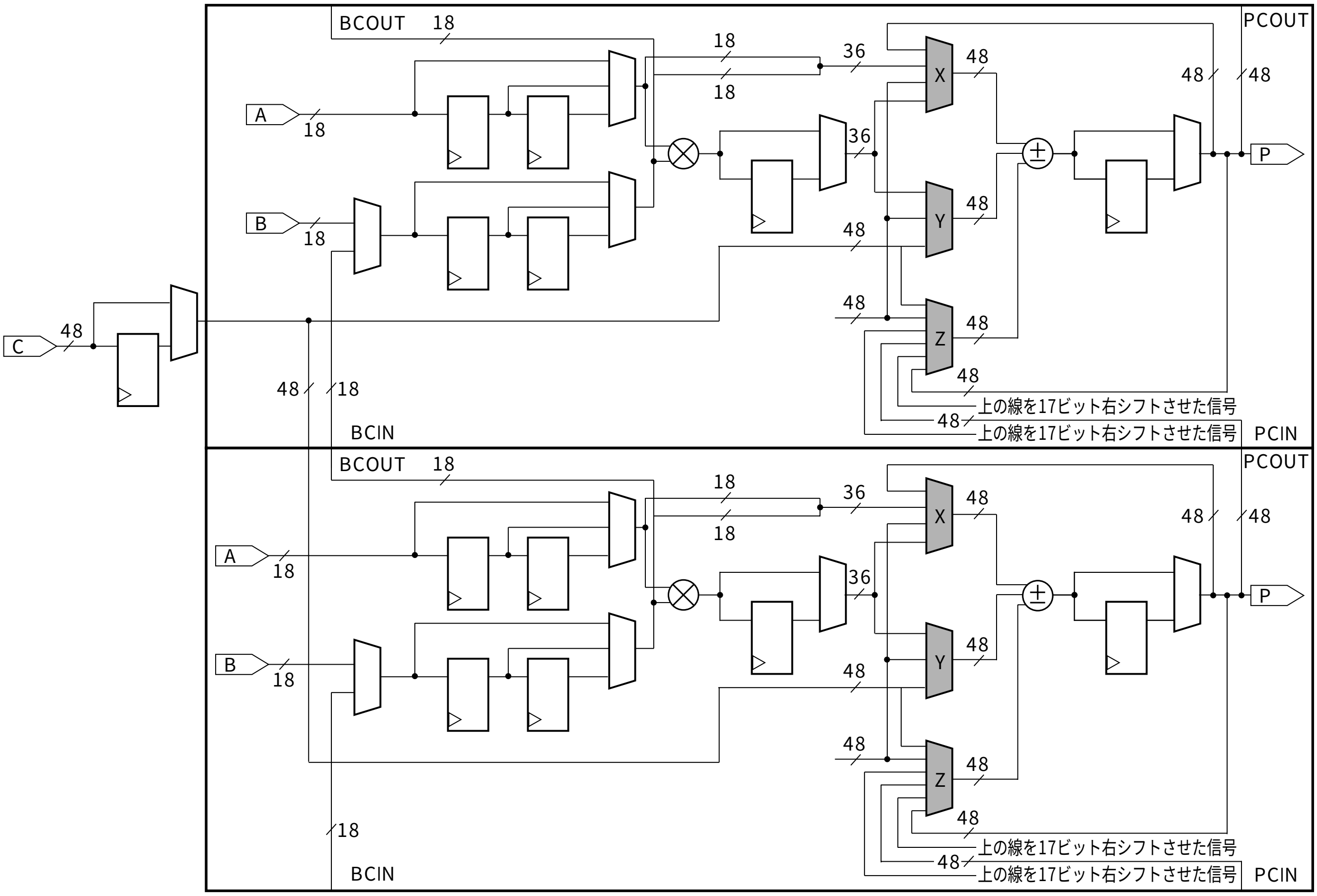


図7　DSP48（2個分）の機能ブロック図
18ビット×18ビットの乗算器に加算器やレジスタを追加したDSPブロックである．X，Y，Zのマルチプレクサの切り替えをクロック・サイクルごとに個別に制御できる．

います．乗算器だけがハード・マクロで搭載されているFPGAで積和演算を行う場合，加算器やマルチプレクサは論理ブロックを使って構成しなければなりません．つまり，論理ブロックによる遅延がボトルネックとなる可能性があります．せっかくの乗算器の能力を活用しきれないということになります．

● 積和演算が可能なDSPブロック

Xilinx社の「Virtex-4ファミリ」では，18ビット×18ビットの乗算器に加算器やレジスタを追加したDSPブロックを搭載しています．

Virtex-4ファミリのDSPブロックを**図7**に示します．**図8**のように，二つのDSP48スライスが並んだ形で配置されています．こうすることによって，互いの演算結果を高速に伝達することができます．例えば，複素数演算や多ビット演算に効果的な配置といえます．

図8
DSP48の配置関係図
DSP48を2タイル並べて構成することで，複素数演算などを効率的に実行することができる．



DSP48では加算器やいくつかの演算モードを切り替えるためのマルチプレクサ，およびレジスタが用意されています．灰色で示したマルチプレクサは，論理合成時にどの入力を使用するかをあらかじめ選択します．動作モードを使用すると，**図7**のX，Y，Zのマルチプレクサの切り替えをクロック・サイクルごとに個別に制御できます．この機構を利用することで，さまざまな演算をクロック・サイクル単位に制御することが可能となっています．そうすれば，ディジタル信号処理アプリケーションの核となる演算部分を余分な論理ブロックを消費せずに実現できます．FPGA

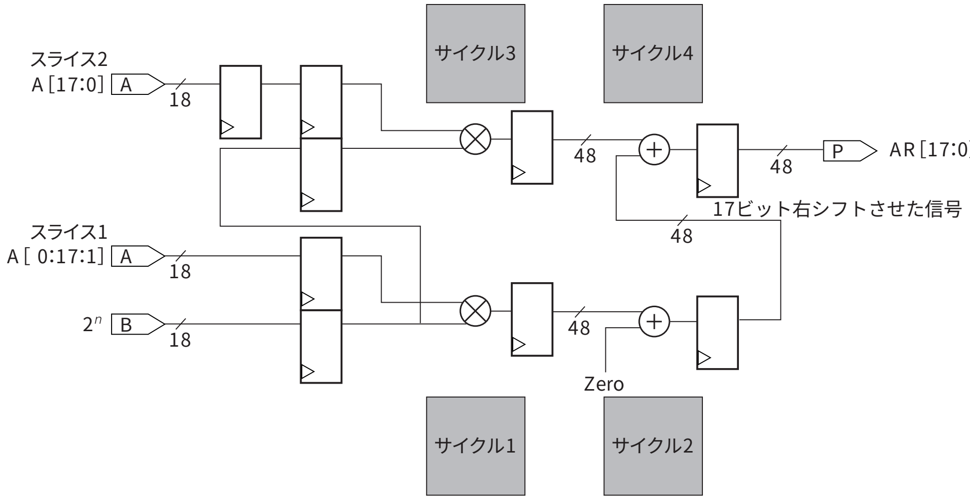


図9
DSPブロックを使ったバレル・シフタ
図7からバレル・シフタで使用するブロックだけを抽出したもの．

向けCPUコアと組み合わせれば，オリジナルのディジタル信号処理プロセッサも実現できます．動作モードの切り替えだけをPicoBlazeのような小型のCPUコアで制御する使いかたも考えられます．

● DSPブロックを使ったバレル・シフタ

DSP48を2個使用して18ビットのバレル・シフタを構成した例[5]を挙げながら，DSP48の使いかたを説明します．**図7**からバレル・シフタで使用するブロックだけを抽出すると**図9**のようになります．シフト動作は，以下のようになります．

1）スライス1を使って入力データに2^nを掛ける．スライス1の入力データは，1ビット右シフトしたもの．
2）スライス1を使ってゼロを加算．演算結果は変わらないが，17ビット右シフトした信号が出力される．
3）スライス3を使って入力データに2^nを掛ける．
4）スライス2を使って2）の出力と加算を行う．

これで下位の18ビットを切り取ったものがシフト結果になっています．

1回のシフトに4サイクル必要ですが，パイプライン構成なので，連続したデータであれば1クロックごとに出力を得ることができます．また，論理ブロックで構成した場合と比べて高速に動作します．

まとめ

本稿では，FPGAに搭載された乗算器を使うためのHDL記述とツールの使いかたについて説明しました．一つのFPGAに多数の乗算器が搭載されているので，まずは＊記号をどんどん使ってみるとよいでしょう．そして，最適化の必要性がでてきたらCORE Generatorなどを活用していくとよいと思います．乗算器を使用することで，複雑なアルゴリズムでも性能を犠牲にすることなく実現できる可能性があります．CPUコアとの組み合わせもお勧めです．

参考・引用＊文献

(1) Xilinx社のホームページ，http://www.xilinx.co.jp/
(2) Spartan-3 FPGAファミリ ： 製品紹介および注文情報，Xilinx，2004年7月．
(3) Altera社のホームページ，http://www.altera.co.jp/
(4) Cyclone II Device Famiry Data Sheet，Altera，Nov. 2004.
(5) XtreameDSP Design Considerations User Guide，Xilinx，Sept. 2004.

たなか・りょうへい
(株)ダイヘン
tanaka@advanced-ip.jp

<筆者プロフィール>
田中良平．今年で34歳．同志社大学工学部 電気工学科卒．ディジタル信号処理アルゴリズムの製品化に際し，FPGAを使って効果的に実現する方式を模索している．